**Отчет о проекте «Sunflower Spaniel»**

**Версия 0.4**

**29 июня 2017 г.**

**Андрей Щипило (**[**andrewshipilo@mail.ru**](mailto:andrewshipilo@mail.ru)**)**

1. **Вступление**

Летом прошлого года мне было предложено поучаствовать в менторской программе. Недолго думая, я решил, что это достаточно интересная возможность и стоит ей воспользоваться. Для того, чтобы попробовать поучаствовать в программе необходимо было написать некое эссе о себе, что я и сделал одним вечером и отправил на рассмотрение. В начале весны пришел ответ, в котором говорилось, что я прошел.

Спустя некоторое время я познакомился с моим ментором – Василием Толстым ([vasiliy.tolstoy@dell.com](about:blank)). Мы обсудили многие вещи, рассказали друг другу об интересах и попытались найти общие. После пары совещаний мы решили, что стоит попробовать сделать движок/игру на C++, OpenGL, GNU Linux, так как мне всегда было интересно, как устроены игровые движки и игры внутри.

1. **Задача**

Сначала основной задачей было создание движка для игр, который можно будет использовать в последующих проектах (<https://github.com/walker2/Falcon>). После определения основной задачи следовало изучить необходимые материалы. Именно для этих целей Василий подготовил обширный список необходимых книг. Больше всего мне пригодились книги Game Engine Architecture и OpenGL Programming Guide (см. раздел «Ссылки»).

Достаточно быстро был написан основной костяк движка, способный достаточно хорошо отрисовывать картинку. Для проверки этого движка/библиотеки необходимо было добавить некую игру, в которой тестировались основные возможности движка. Мой выбор пал на игру типа Shoot’em up, где, по замыслу, нужно было отстреливаться от зомби, защищая беззащитных жителей (<https://github.com/walker2/KillEmAll>).

Далее последовал небольшой тест на производительность и физ. движок (<https://github.com/walker2/BallPit>).

В конце концов было решено сделать игру масштабнее предыдущих, при этом использовать правильную организацию проекта, и для облегчения задачи добавить физ. движок Box2D.

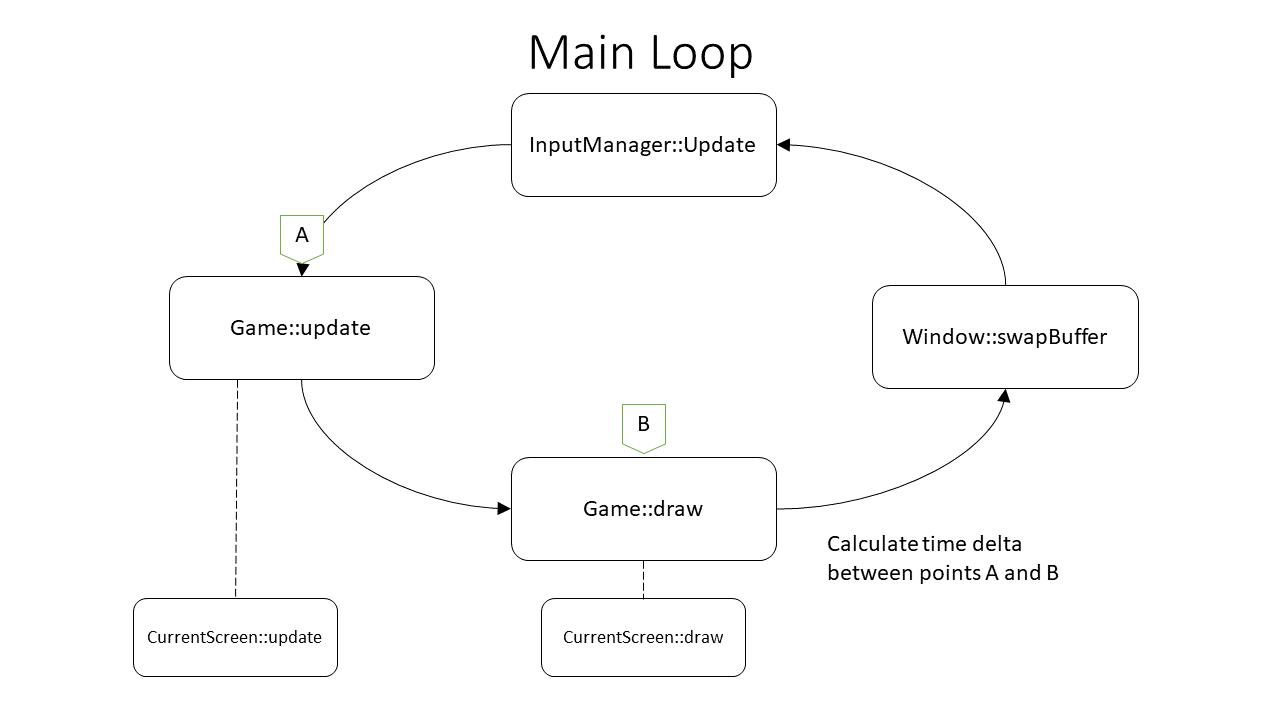
1. **Выбор технологий**

Для разработки мы выбрали следующие технологии:

* C++, так как это основной язык для движков больших студий, потому что он позволяет наиболее плотно оптимизировать игровые процессы, что очень важно, ведь производительность играет высокую роль
* CMake, для оптимизации сборки проектов, что позволило в конечном счете сэкономить достаточно большое количество времени и усилий.
* GNU / Linux, был выбран, так как является открытой платформой с которой был очень хорошо знаком Василий, а я – почти нет.
* OpenGL, очевидно следует из предыдущего пункта, при он способен предоставить хорошую переносимость и на другие платформы.
* Box2D, я выбрал уже, когда стал делать проект SunflowerSpaniel, поняв, что для хорошей оптимизации уже существующей физ. системы потребовалось огромное количество усилий.

1. **Общая структура игровой машины**

Как и в любом игровом движке. Основой всего стоит так называемый Main(Game) Loop, который поочередно вызывает обновление, отрисовку и опрашивает систему на счёт ввода пользователя. Укрупненное представление Main Loop представлено на Рис. 1.



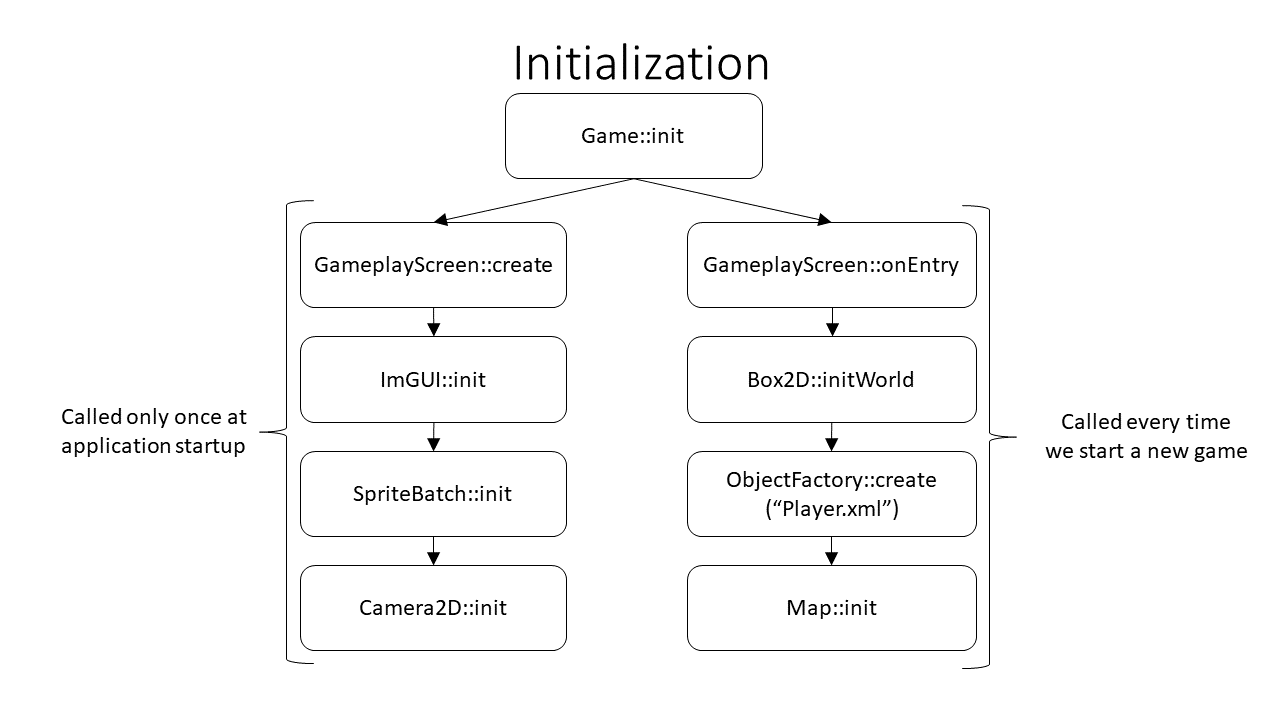
*Рис. 1 Main Loop*

В классе IMainGame в методе run() идет бесконечный цикл. Высчитывается нынешнее значение totalDeltaTime, получаемое из разницы между предыдущим и нынешним значением количества тиков. Опрашивается InputManager вызовом функции update(), в которой проверяются нажатые клавиши. Начинается второй цикл, в котором вызываются методы IMainGame:

update(), который переключается между различными экранами, присваивая m\_currentScreen необходимый экран, и вызывает у m\_currentScreen метод update().

draw(), который вызывает метод draw() у m\_currentScreen, отвечающий за отрисовку экрана.

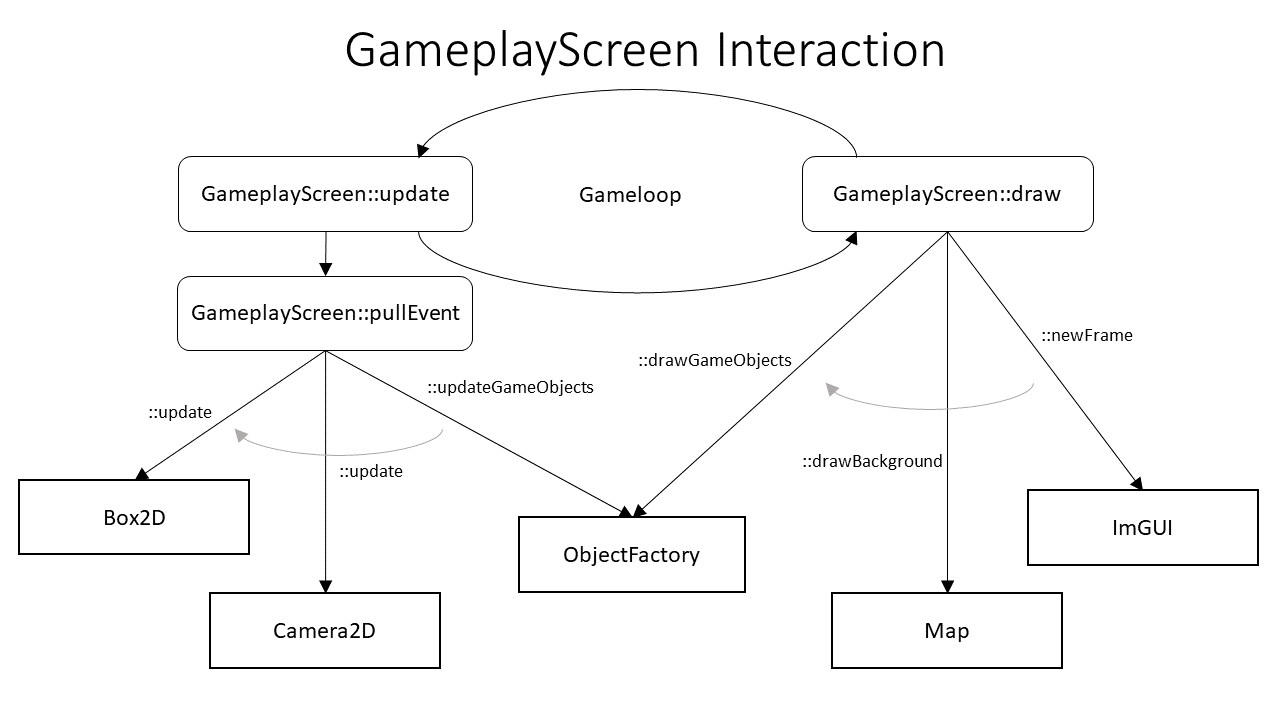
Так как нам необходимо переключаться между экранами, важно, чтобы инициализация игры и экранов происходила корректно, выделяя нужную и освобождая ненужную память. Поэтому инициализация разделена на две части: инициализация, происходящая при создании объекта, и инициализация, которая вызывается при переключении экрана. Две ветви инициализации представлены на Рис. 2.



*Рис. 2 Инициализация*

Так, например, в методе create() в классе GameplayScreen, происходит инициализация рендера, компилируются шейдеры, устанавливаются границы камеры и т.д., а в методе onEntry() мы создаем новый Box2D world, в котором хранятся все данные о физике наших объектов, создаем двух игроков (собаку и человека), а также генерируем случайную карту, заполняя ее объектами в методе generateMap класса Map.

Далее, стоит вернуться к методам Screen: update() и draw(). Именно из этих методов происходит вся основная интеракция и вызовы методов других классов. Диаграмма основных взаимодействий класса GameplayScreen приведена на Рис. 3.



*Рис. 3 Взаимодействия GameplayScreen*

Рассмотрим метод update():

В начале мы вызываем метод pullEvent, который позволяет принимать нам любые события, которые нам отдаёт система, т. е. благодаря именно этому методу мы можем как либо взаимодействовать и обновлять наше приложение.

Следом идёт вызов метода updateGameObjects класса ObjectFactory. Он проходит по коллекции всех объектов и вызывает метод update() у каждого объекта.

Затем мы высчитываем позицию камеры, которая чаще всего равна позиции игрока и обновляем камеру, вызывая метод update().

Далее мы обновляем Box2D world, вызывая метод Step(), который обновляет позиции всех тел в физической симуляции движка.

В конце update() мы удаляем все объекты, которые были добавлены в очередь на удаление за этот цикл, вызывая deleteGameObjects() у класса ObjectFactory.

Перейдем к методу draw():

Метод draw() отрисовывает элементы на экране. Сперва нам необходимо создать новый кадр GUI библиотекой – imgui. Далее мы очищаем и экран и подготавливаем его для отрисовки функциями из OpenGL. Создаем uniform текстур, цвета, а также получаем матрицу камеры из класса Camera.

Так как в рамках одного прохода функции мы отрисовываем в поверх предыдущего, нам необходимо первым делом нарисовать задний фон, мы его получаем из Map методом drawBackground(). Но важно заключить отрисовку текстур с использованием шейдеров предназначенных для рисования текстур, т.е. m\_textureProgram, в который мы уже скомпилировали необходимые нам шейдеры. Теперь достаточно поместить drawBackground() между вызовами методов use() и unuse().

|  |  |
| --- | --- |
|  | m\_textureProgram.unuse(); |
|  | if (m\_map.isGenerated()) |
|  | { |
|  | m\_map.drawBackground(); |
|  | } |
|  |  |
|  | m\_textureProgram.unuse(); |

То же самое следует сделать и для отрисовки света, но так как нам нужно полотно (Batch), куда мы будем его отрисовывать необходимо использовать конструкцию batch.begin(), batch.end(), batch.renderBatch()

|  |  |
| --- | --- |
|  | m\_spriteBatch.begin(); |
|  | playerLight.draw(m\_spriteBatch); |
|  | mouseLight.draw(m\_spriteBatch); |
|  | m\_spriteBatch.end(); |
|  | m\_spriteBatch.renderBatch(); |

SpriteBatch::begin() – очищает массив RenderBatches для дальнейшего использования.

SpriteBatch::end() – создает массив RenderBatches из массива Glyphs, в который записываются глифы, добавляемые в методе SpriteBatch::draw() и сортирует его по textureID

SpriteBatch::draw() – добавляет в массив новый Glyph, в который входит textureID, глубина и размеры в виде координат вершин прямоугольника.

SpriteBatch::renderBatch() – привязывает текстуры по textureID, указывает параметры отрисовки и рисует массивы вершин.

После того как мы отрисовали свет следует отрисовка всех игровых объектов через функцию класса ObjectFactory drawGameObjects(), в которой мы проходим по всем объектам, выбираем те у которых есть компоненты для отрисовки, добавляем их в массив, сортируем массив по Z компоненте положения в пространстве и отрисовываем необходимые компоненты, вызывая соответствующие функции (прим. spriteComponent->draw(spriteBatch);)

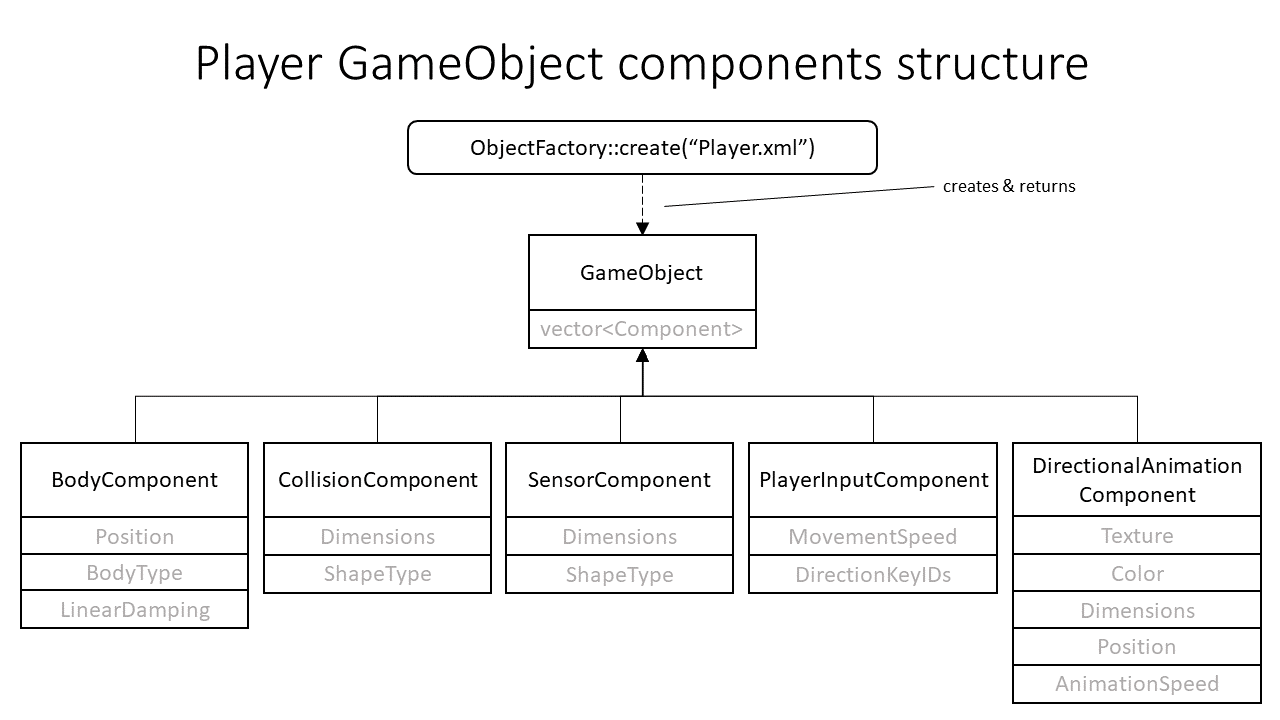
Далее подобным образом отрисовываются foreground из Map, а также debug GUI и GUI инвентаря.

Стоит теперь рассмотреть большую и основную часть реализации – это архитектурный паттерн Entity-Component-System, который использует композицию вместо огромных деревьев наследования.

В основе лежит понимание, что entity – это некий пустой сосуд, который можно заполнить необходимыми компонентами, а затем проверять подходит ли его наполнения нашим системам, которые используют данные из компонентов в своих целях.

В моей реализация ECS нет яркого выделения одной из компонент – System, так как мне в своё время показалось более удобным сосредоточить весь смысл именно в Component. Следовательно, большую часть кода, чаще всего присущего таким системам, можно обнаружить в Component. В роли «систем» выступают отдельные функции, которые проверяют необходимость обработки того или иного Entity, в соответствии с наличием тех или иных компонентов. (Например, см. метод ObjectFactory::drawGameObjects()).

Любая Entity в моей реализации создается внутри метода ObjectFactory::createObject(), которому на вход подается .XML файл с описанием компонентов того или иного объекта. Затем файл просматривается, и из него создаются и добавляются компоненты с соответствующими параметрами. Обычно это происходит внутри класса компонента в методе init, который устанавливает необходимые параметры в поля класса.



*Рис. 4 Устройство GameObject*

Таким образом, для создания некого уникального объекта нам необходимо создать .xml файл с описанием нужных компонентов.

Для регистрации нового компонента нужно отнаследовать его от Component и добавить в функцию addComponent() поведение при нахождении его имени при просмотре xml-файла, а затем использовать его особенности и методы в одной из функций.

Для того, чтобы получить необходимый компонент из любого entity (GameObject), нужно вызвать метод GameObject::getComponent<T>():

**auto** playerPos = m\_currentPlayer->getComponent<BodyComponent>()->getPosition();

1. **Описания геймплея**

*Здесь описываем геймплей в общих словах, но достаточно подробно. Он определяет выбор технических решений, так что его описание нельзя отнести в самый конец.*

1. **Детали реализации**

В процессе работы над этим проектом возникало огромное количество вопросов, но одним из самых интересных оказался вопрос генерации приятного глазу ландшафта.

С самого начала нужно было случайно сгенерировать карту. Немного покопавшись в интернете, я нашел занятную метод генерации матрицы случайных значений – Шум Перлина. Это достаточно популярный метод, и я быстро нашёл отдельную функцию для генерации для конкретных значений x и y.

elevation[y][x] = noise(nx, ny);

Однако такой подход выглядит достаточно скучно даже в виде матрицы с значениями от 0.0 до 1.0, не говоря уже о графике. Поэкспериментировав с разными значениями и почитав про шумы в интернете стало очевидно, что можно изменять значения перед функцией и аргументами, регулируя тем самым подъем и частоту генерации шума.

В конце концов я пришел к такому виду функции:

|  |  |
| --- | --- |
|  | **double** e = (1.00 \* noise(1 \* nx, 1 \* ny)  + 0.50 \* noise(2 \* nx, 2 \* ny)  + 0.25 \* noise(4 \* nx, 4 \* ny)  + 0.13 \* noise(8 \* nx, 8 \* ny)  + 0.06 \* noise(16 \* nx, 16 \* ny)  + 0.03 \* noise(32 \* nx, 32 \* ny));  e /= (1.00 + 0.50 + 0.25 + 0.13);  m\_value[x][y] = e; |

Далее достаточно быстро стало понятно, что нужно распределить значения по неким областям, биомам. С этим я достаточно долго экспериментировал потом ещё, но остановился на следующем распределении:

|  |
| --- |
|  |
|  | **if** (val == -1) **return** BIOME::ROAD; |
|  | **else** **if** (val <= 0.15) **return** BIOME::WATER; |
|  | **else** **if** (val <= 0.25) **return** BIOME::BEACH; |
|  | **else** **if** (val <= 0.55) **return** BIOME::FOREST; |
|  | **else** **if** (val <= 0.65) **return** BIOME::JUNGLE; |
|  | **else** **if** (val <= 0.75) **return** BIOME::SAVANNAH; |
|  | **else** **if** (val <= 0.85) **return** BIOME::ROCKS; |
|  | **else** **return** BIOME::SNOW; |

Потом нужно было каждому биому поставить в соответствие свой спрайт, а затем отрисовать их на экран, что достаточно просто.

На этот момент всё выглядело примерно так:

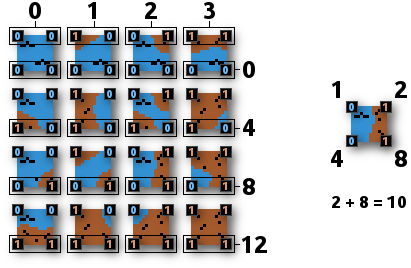


*Рис. 5 Тайлы без переходов*

Трудно не согласиться, что это выглядит, в лучшем случае, приемлемо. Однако, было очевидно, что были необходимы некоторые переходные спрайты между биомами.

В процессе поиска я перепробовал многие методы, допустим я долго провозился с идеей убирать альфа-канал у спрайтов и накладывать друг на друга, но такой подход хоть и выглядел чуть лучше, но всё равно не устраивал меня. Через несколько дней поисков я наткнулся на отличную статью, объясняющую применение алгоритма компьютерной графики Marching Squares (с англ. — «движущиеся квадраты»): <http://blog.project-retrograde.com/2013/05/marching-squares/>.

Я взял оттуда основную идею, однако высчитывать tileID через седловые точки кольца показалось мне не слишком понятной реализацией, хоть и весьма элегантной, и я высчитывал tileID в паре if.



*Рис. 6 Пример Marching Squares*

Получив функцию, вычисляющую нужный нам тайл по соседним тайлам, мне нужно было только нарисовать все спрайты и расположить их в определенном порядке:



*Рис. 7 Фрагмент файла с переходными спрайтами*

Пример вычисления индекса по 8 ближайшим тайлам:

**if** (sTL != trans && sT != trans && sL != trans)

{

index += 1;

}

**if** (sT != trans && sTR != trans && sR != trans)

{

index += 2;

}

**if** (sL != trans && sBL != trans && sB != trans)

{

index += 4;

}

**if** (sR != trans && sBR != trans && sB != trans)

{

index += 8;

}

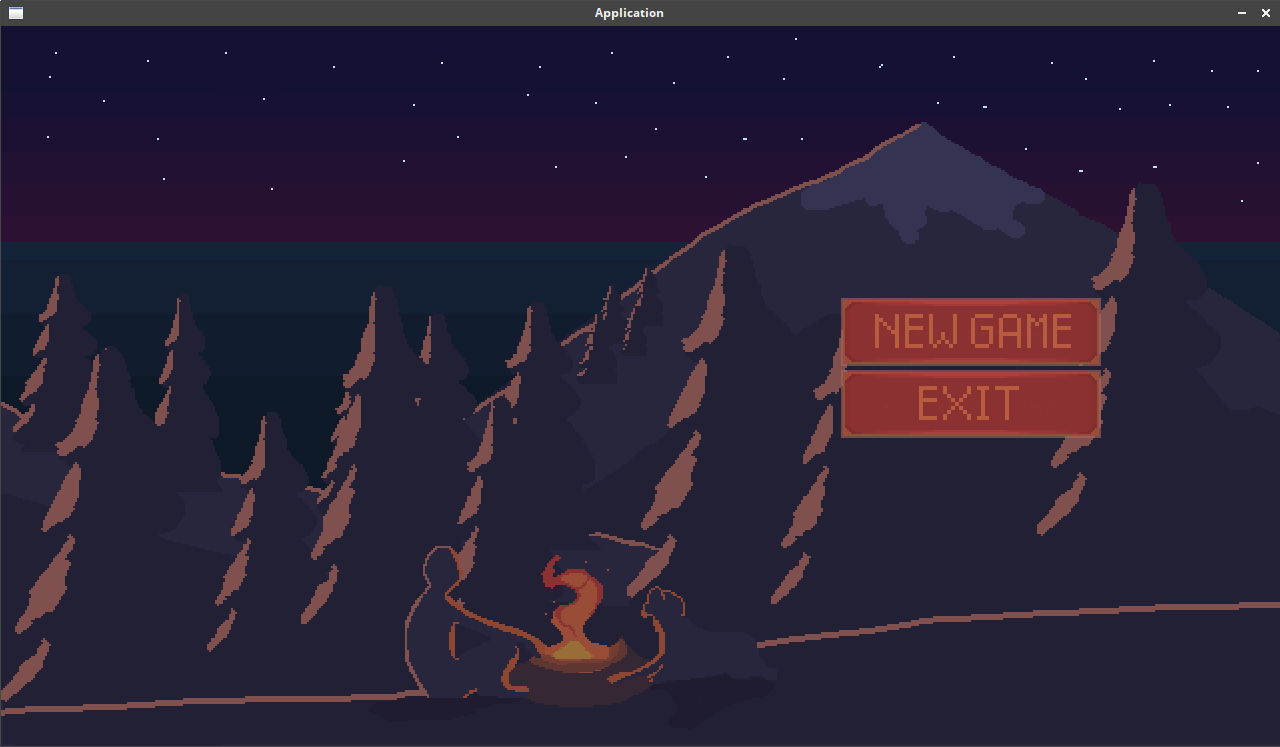
Осталось добавить несколько деревьев и разнообразить сами тайлы земли, и мы получаем вполне органично выглядящую картинку:



*Рис. 8 Тайлы с переходами*

1. **Состояние проекта на сегодня**

При запуске программы на экране появляется главное меню, с двумя достаточно очевидными кнопками: new game и exit. В дальнейшем необходимо будет добавить кнопку settings для изменения настроек игры, будь то громкость, разрешение и т.д.

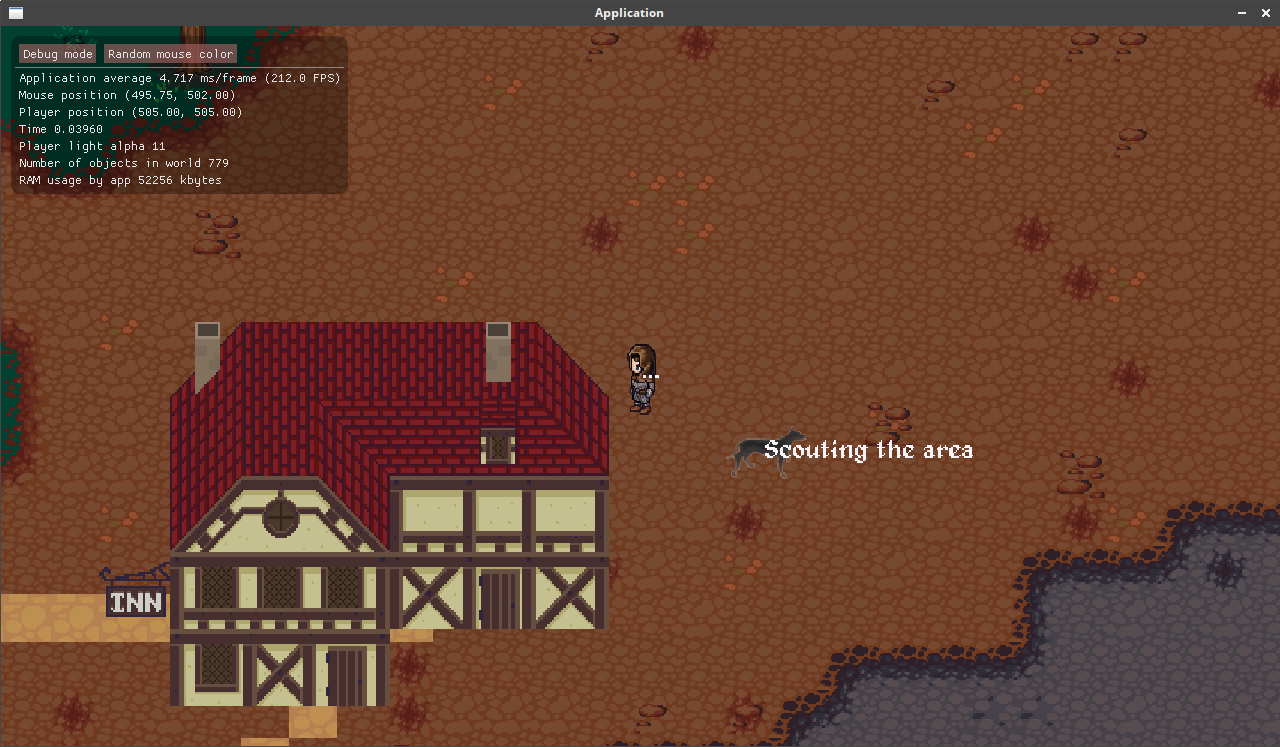


*Рис. 9 Главное меню*

При нажатии на new game мир генерируется с нуля, персонажи помещаются в определенное место. Появляются в указанных в xml местах дома и NPC, между ними прокладывается путь. Скорее всего можно дать доступ пользователю к различным настройкам генерации, позволив изменить кол-во воды, различных зон, количества зданий и NPC, и так далее. Также достаточно важным улучшением будет загрузка и сохранение текущего состояния игры.

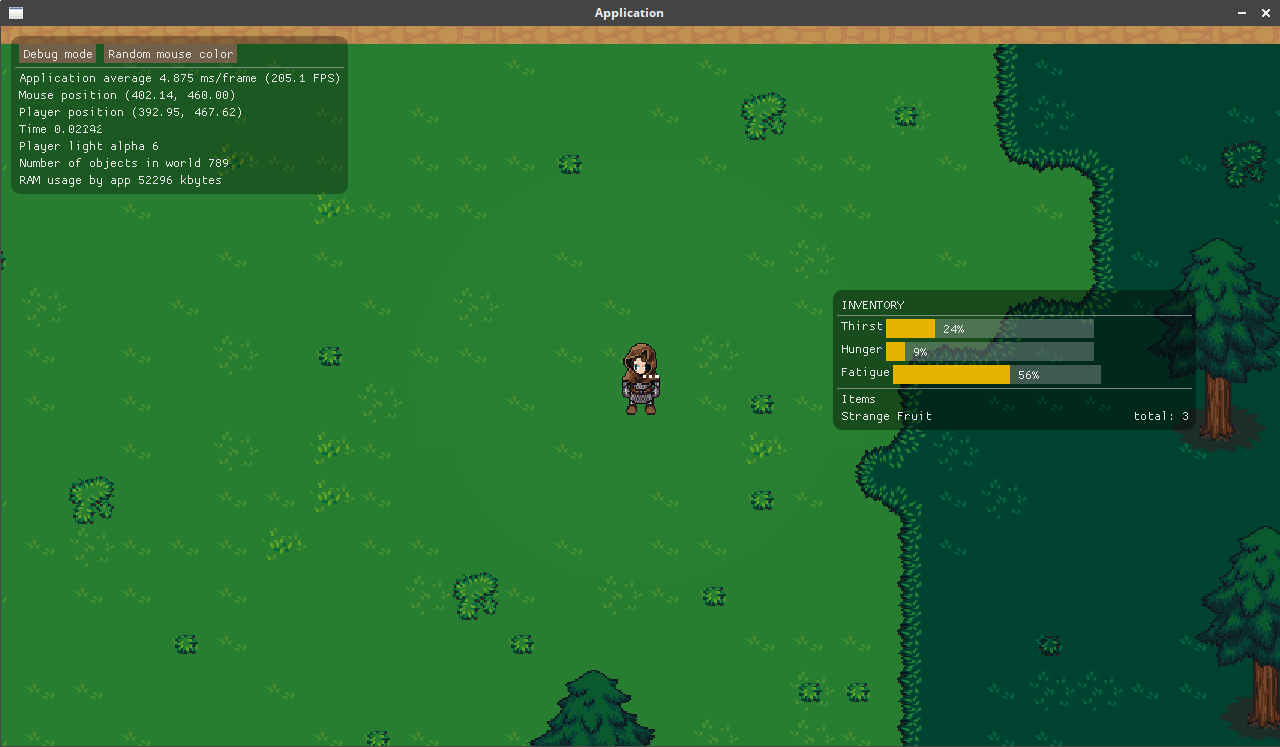
Одной из нерешенных проблем является долгая генерация мира при нажатии на кнопку new game. Есть множество способов решения данной проблемы, одна из которых – это ускорение генерации объектов в мире в методе ObjectFactory::createObject.

При успешной загрузке на экране появляются главные персонажи, между которыми можно переключаться на клавишу TAB. При нажатии клавиши ESC будет загружено главное меню и предоставлен возможность сгенерировать мир заново. За передвижение отвечают клавиши W,A,S,D.



*Рис. 10 Игровой экран после генерации мира*

При нажатии клавиши I за любого персонажа будет открыто окно инвентаря соответствующего персонажа, на котором будут показаны жажда, голод и усталость.



*Рис. 11 Окно инвентаря*

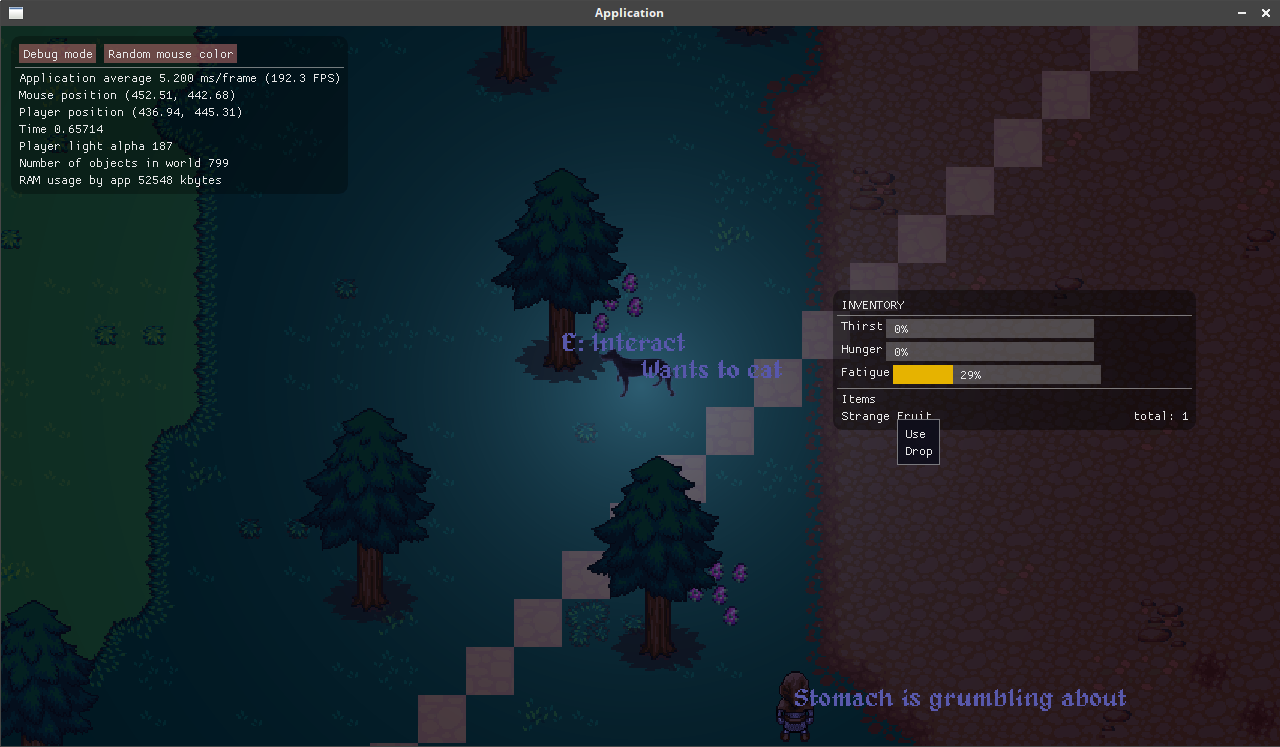
Нынешнее состояние персонажа рисуется поверх него текстом, кратко описывающим его. Так, например, персонаж собаки может спать или охотится. Сейчас состояния никак не влияют на персонажей, это необходимо будет доработать и развить, так как это одна из главных основ геймплея.

При приближении к интерактивным объектам появляется текст, подсказывающий клавиши для интеракции. Так, например, на клавишу E мы можем взаимодействовать с деревом, с которого собьём плоды, или же можем поднять плод с земли и положить его в инвентарь.



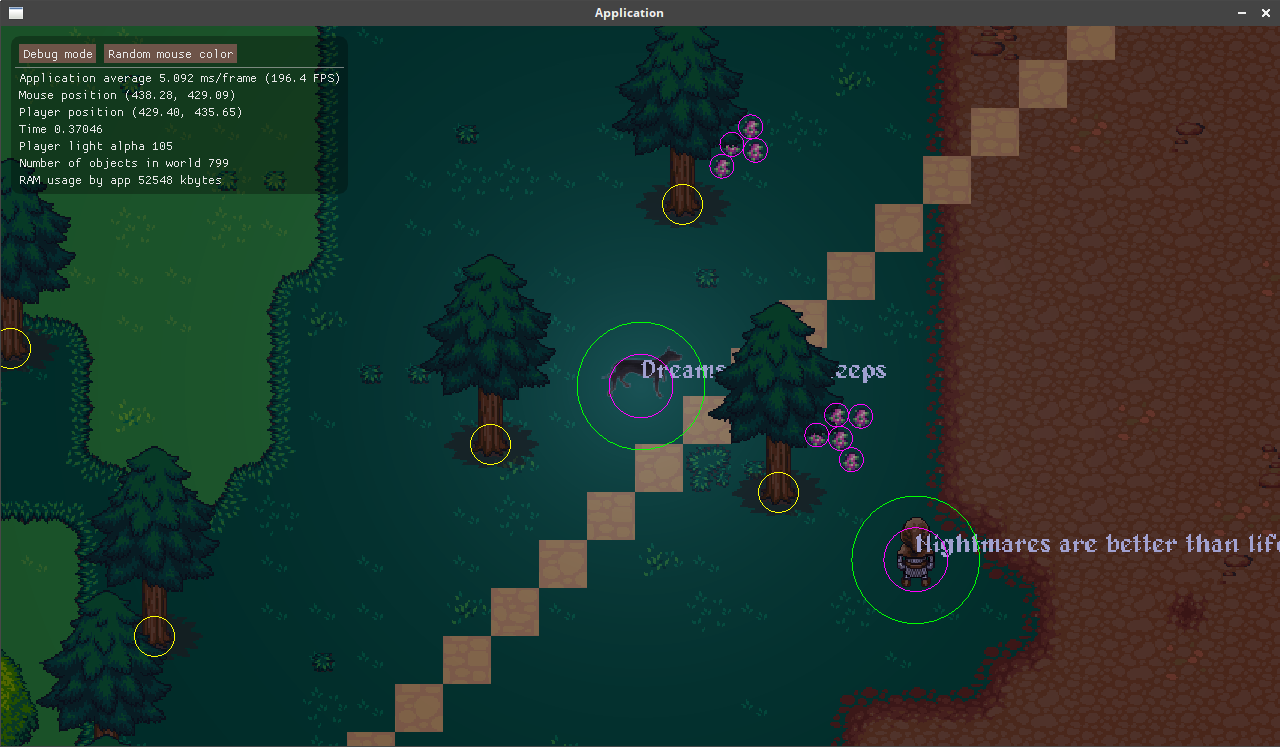
*Рис. 12 Интеракция с объектами мира*

После поднятия предмета он перемещается в инвентарь, где мы можем на нем нажать ПКМ и выбрать: съесть его или выбросить.



*Рис. 13 Использования предметов в инвентаре*

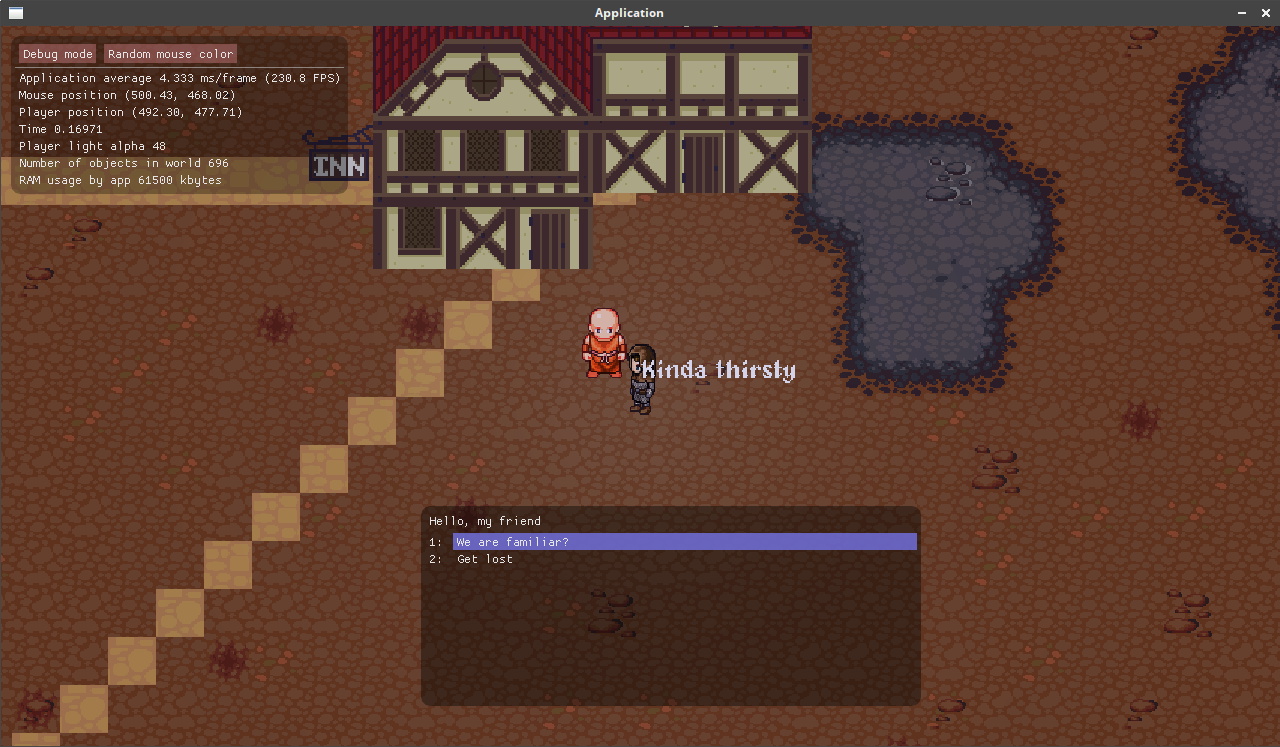
При желании можно нажать на кнопку в верхнем углу Debug mode, расположившуюся выше полезной информации вроде количества кадров в секунду, позиции игрока и мышки, использованного объёма памяти и т.д. Кнопка debug mode показывает нам границы физических тел в мире, при этом желтая граница – статические тела, фиолетовая – динамические, а зеленая – зона взаимодействия с другими тела.



*Рис. 14 Debug mode*

На Рис. 14 показан ещё один небольшой баг, а именно — текст рисуется на слое с спрайтом персонажа, поэтому он может заходить за объекты, которые находятся впереди персонажа.

При взаимодействии с неигровыми персонажами перед нами возникает окошко диалога, где мы можем выбирать с помощью мышки варианты ответа



*Рис. 15 Окно диалога*

Данную систему можно расширять добавлять варианты ответов и вопросов, и разветвлять диалоги так, чтобы разные ответы приносили разные результаты.

Ещё одной, пока не разрешенной, проблемой является возможность генерации определенных объектов на разных этапах дороги, будь то деревья, более сложные звери или квестовые здания/объекты. Такая осложненная генерация определенно представляет собой неординарную и интересную задачу.

Как видно из вышесказанного, сделана достаточно большая часть работы, но и немалая часть лежит впереди, нужно будет использовать доступные системы, всячески их улучшая и расширяя, добавляя новые возможности.

1. **Перспективы развития**

На данный момент проект лежит в открытом доступе по адресу <https://github.com/walker2/SunflowerSpaniel>.

Для завершения проекта необходимо проделать множество работы, поэтому предлагаю всем желающим присоединиться к нему. Впрочем, даже если не хочется продолжать делать чей-то чужой проект и разбираться в тонкостях реализации, то всегда можно посмотреть на общую структуру или на решение каких-то определенных проблем на живом примере.

Сейчас существует возможность сборки проекта только на Linux, с установкой необходимых библиотек. Однако, CMake – это кроссплатформенный инструмент, и с достаточным знанием, скорее всего, не потребуется больших затрат для того, чтобы запустить проект на других системах.

1. **Ссылки и литература**

*Широкой рукой ссылки на всё: статьи, сайты, использованные книги.*

1. **Приложения**

*Сюда можно поместить все материалы по сюжету игры со скетчами и примерами диалогов. Наверное, здесь же – неиспользованные диаграммы. Можно повторить все диаграммы подряд, если есть смысл, и, если хочется – подробнее прокомментировать.*